Python Handout

English edition

Contents

1 Getting started **3**
- 1.1 Your first program . . . 3
- 1.2 Comments & code style . . . 3
- 1.3 Input, process, output . . . 3

2 Variables, types & operators **4**
- 2.1 Variables & assignment . . . 4
- 2.2 Numbers: int & float . . . 4
- 2.3 Expressions & type conversion . . . 5
- 2.4 Booleans & comparison . . . 5

3 Strings **6**
- 3.1 Indexing . . . 6
- 3.2 Slicing . . . 6
- 3.3 String methods & length . . . 6
- 3.4 f-strings . . . 7

4 Selection **8**
- 4.1 if / elif / else . . . 8
- 4.2 Combining conditions . . . 8

5 Iteration **9**
- 5.1 for loops and range . . . 9
- 5.2 The accumulator pattern . . . 9
- 5.3 while loops . . . 9
- 5.4 Nested loops . . . 10

6 Lists & 2-D lists **11**
- 6.1 Lists . . . 11
- 6.2 Traversing a list . . . 11
- 6.3 2-D lists (grids) . . . 11
- 6.4 List comprehensions . . . 12

7 Dictionaries **13**
- 7.1 Dictionaries . . . 13

8 Functions & abstraction **14**
- 8.1 Defining & calling functions . . . 14
- 8.2 Return values . . . 14
- 8.3 Parameters, arguments & scope . . . 14
- 8.4 Procedural abstraction . . . 14
- 8.5 Modules & imports . . . 15

9 Errors, exceptions & testing **16**
- 9.1 Errors & debugging . . . 16
- 9.2 try / except / raise . . . 16
- 9.3 Testing & robustness . . . 16

10 Files **18**
10.1 Reading & writing text files 18

11 Algorithm design **20**
11.1 Algorithms & decomposition 20
11.2 Pseudocode & flowcharts 20
11.3 Recursion & the call stack 20

12 Data structures **22**
12.1 Abstract Data Types (ADTs) 22
12.2 Stacks 22
12.3 Queues 22
12.4 Linked lists 23
12.5 Hash tables 23
12.6 Binary search trees 24
12.7 Graphs 24

13 Searching, sorting & efficiency **26**
13.1 Linear & binary search 26
13.2 Sorting (bubble & insertion) 26
13.3 Algorithmic efficiency 27
13.4 Randomness & simulation 27

14 OOP & paradigms **28**
14.1 Classes & objects 28
14.2 Inheritance, encapsulation & polymorphism 28
14.3 Programming paradigms 29

15 Data representation **30**
15.1 Bits & binary 30
15.2 Compression 30

16 Computing concepts **32**
16.1 What computing is & the design cycle 32
16.2 The Internet 32
16.3 Parallel & distributed computing 33
16.4 Impact of computing 33

17 Putting it together **34**
17.1 End-to-end mini-projects 34

1 Getting started

1.1 Your first program

Python runs your code one line at a time. Each line is a statement 语句. A program 程序 is just a list of statements that run from top to bottom.

The `print()` function shows text on the screen. This is called output 输出. Text inside quotes is a string 字符串.

```
print("Hello, world!")
print("I am learning Python")
```

- Each `print()` starts a new line.
- Quotes can be `"double"` or `'single'` —both make a string.
- A program does nothing until you run it.

1.2 Comments & code style

A comment 注释 starts with `#`. Python ignores everything after the `#` on that line. Comments explain your code to people; they do not change what the code does.

```
# This line is a note for humans
print("Hi")        # you can also comment at the end of a line
```

Good style makes code easy to read:

- Use clear names that say what a value means.
- Put one statement on each line.
- Do not add spaces at the start of a normal line. In Python, spacing at the start (indentation 缩进) has a special meaning, so a wrong space gives an error 错误.

1.3 Input, process, output

Many programs follow a simple plan: **input** 输入 → **process** → **output**. You get some data, do something with it, then show a result.

The `input()` function reads text that the user types. It always gives back a string.

```
name = input("What is your name? ")
print("Hello, " + name)
```

- `input()` waits for the user to type and press Enter.
- Store the typed text in a variable 变量 so you can use it later.
- Because `input()` returns a string, change it with `int(...)` first if you need a number.

2 Variables, types & operators

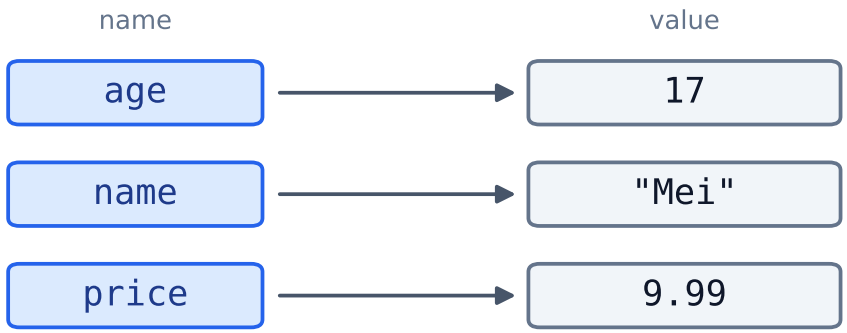
2.1 Variables & assignment

A variable 变量 is a name for a value 值. You make one with `=`, which is called assignment 赋值. The name goes on the left; the value goes on the right.

```
age = 17
name = "Mei"
price = 9.99
print(age, name, price)
```

Now `age` holds `17`. Use the name anywhere you need the value, and change it later:

```
age = 17
age = age + 1   # age is now 18
print(age)
```



Each variable name points to a value in memory

- The `=` sign does not mean "equal". It means "store the right side under the left name".
- To test if two values are equal, use `==` (see below).

2.2 Numbers: int & float

Python has two main number types. An integer 整数 (`int`) is a whole number like `17`. A float 浮点数 (`float`) has a decimal point like `9.99`.

These operators 运算符 work on numbers:

Operator	Meaning	Example	Result
`+`	add	`3 + 2`	`5`
`-`	subtract	`3 - 2`	`1`
`*`	multiply	`3 * 2`	`6`
`/`	divide (always float)	`7 / 2`	`3.5`
`//`	integer divide	`7 // 2`	`3`
`%`	remainder (modulo)	`7 % 2`	`1`
`**`	power	`2 ** 3`	`8`

- `/` always gives a float, so `4 / 2` is `2.0`.
- `//` and `%` go together: `17 // 5` is 3, and `17 % 5` is 2.

2.3 Expressions & type conversion

An expression 表达式 is anything that has a value, like `3 + 4 * 2`. Python uses normal maths order (`*` and `/` before `+` and `-`); add brackets to make the order clear.

`input()` gives a string, so convert it before doing maths. Changing a value from one type to another is type conversion 类型转换:

```
age = int("17")        # text "17" -> number 17
price = float("9.99")  # text -> 9.99
label = str(17)        # number -> text "17"
print(age, price, label)
```

- `int("abc")` fails, so only convert text that looks like a number.
- Mixing types fails too: `"age: " + 17` is an error; write `"age: " + str(17)`.

2.4 Booleans & comparison

A Boolean 布尔值 is one of just two values: `True` or `False`. A comparison 比较 gives back a Boolean.

Operator	Meaning
`==`	equal to
`!=`	not equal to
`< >`	less than / greater than
`<= >=`	less / greater than or equal to

```
print(7 > 2)        # True
print(3 == 3.0)     # True
age = 20
print(age >= 18)    # True
```

Join comparisons with `and`, `or`, `not`:

```
age = 20
print(age >= 13 and age <= 19)   # True only for a teenager
```

3 Strings

3.1 Indexing

A string 字符串 is text inside quotes. Each character 字符 has a position, called its index 索引. The first index is `0`, not `1`.

Read one character with square brackets:

```
word = "Python"
print(word[0])    # P  (the first character)
print(word[2])    # t
print(len(word))  # 6  (how many characters)
```

- Counting starts at `0`, so the last index is `len(word) - 1`.
- A negative index counts back from the end: `word[-1]` is the last character.

```
word = "Python"
print(word[-1])   # n
print(word[-2])   # o
```

- An index that is too large gives an error 错误 (an `IndexError`).

3.2 Slicing

A slice 切片 takes a part of a string. Write `word[start:end]`. The slice keeps `start` but stops **before** `end`.

```
word = "Python"
print(word[0:3])   # Pyt   (positions 0, 1, 2)
print(word[2:5])   # tho
```

- Leave out `start` to begin at `0`; leave out `end` to go to the end.

```
word = "Python"
print(word[:3])    # Pyt
print(word[3:])    # hon
```

- A third number is the step 步长. `word[::-1]` reverses 反转 the string.

```
print("Python"[::-1])   # nohtyP
```

3.3 String methods & length

A method 方法 is a function that belongs to a value. You call it with a dot:

```
name = "mei chen"
print(name.upper())     # MEI CHEN
print(name.title())     # Mei Chen
print(len(name))        # 8
```

Strings are immutable 不可变: a method returns a **new** string and never changes the original 原始 one.

```
name = "mei"
print(name.upper())     # MEI  (the returned value)
print(name)             # mei  (the original is unchanged)
```

Common methods (each returns a new value):

Method	Meaning	Example	Result
`.upper()` / `.lower()`	change case	`"Hi".lower()`	`hi`
`.strip()`	remove edge spaces	`" hi ".strip()`	`hi`
`.replace(a, b)`	swap text	`"cat".replace("c", "b")`	`bat`
`.split(sep)`	break into a list	`"a,b".split(",")`	`['a', 'b']`

Join strings with `+`. This is called concatenation 拼接:

```
first = "Mei"
last = "Chen"
print(first + " " + last)   # Mei Chen
```

3.4 f-strings

An f-string 格式化字符串 builds text from values. Put `f` before the quote, then write `{...}` around a value.

```
name = "Mei"
age = 17
print(f"{name} is {age} years old")   # Mei is 17 years old
```

- Any expression 表达式 can go inside the braces.
- `{value:.2f}` rounds to 2 decimal places 小数位.

```
price = 9.5
print(f"Two cost {price * 2}")        # Two cost 19.0
print(f"Pi is about {3.14159:.2f}")   # Pi is about 3.14
```

4 Selection

4.1 if / elif / else

A program chooses what to do with `if`. It runs an indented 缩进 block only when a condition 条件 is true. The `if` line ends with a colon 冒号.

```
score = 72
if score >= 60:
    print("pass")
# pass
```

Add `elif` (else-if) for more cases and `else` for "anything else". Python runs the **first** true branch 分支 only, then skips the rest.

```
score = 72
if score >= 80:
    print("A")
elif score >= 60:
    print("B")
else:
    print("fail")
# B
```

- Compare values with `==` (equal to), `!=` (not equal to), `<`, `>`, `<=`, `>=`.
- A comparison 比较 gives a Boolean 布尔值—either `True` or `False`.

4.2 Combining conditions

Join conditions with `and`, `or`, `not`. `and` needs **both** sides true; `or` needs **either** side true; `not` flips a Boolean.

```
age = 16
has_ticket = True
if age >= 18 and has_ticket:
    print("entry allowed")
else:
    print("entry refused")
# entry refused
```

- Use brackets to make the order clear: `(a or b) and c`.

```
temp = 30
if temp > 25 and not temp > 35:
    print("warm but ok")
# warm but ok
```

5 Iteration

5.1 for loops and range

A loop 循环 repeats code. A `for` loop repeats once for each item in a sequence 序列. `range(n)` gives the numbers `0` up to `n - 1`.

```
for i in range(5):
    print(i)
# 0, then 1, 2, 3, 4 (each on its own line)
```

- `range(a, b)` goes from `a` up to (but not including) `b`.
- `range(a, b, step)` adds a step 步长 each time.

```
for n in range(2, 11, 2):
    print(n)        # 2 4 6 8 10
```

5.2 The accumulator pattern

To build a result across a loop, start a variable **before** the loop, then update 更新 it each turn. This is the accumulator 累加器 pattern.

```
total = 0
for n in range(1, 6):
    total = total + n
print(total)        # 15
```

- The same idea counts how many items match a test.

```
count = 0
for letter in "banana":
    if letter == "a":
        count = count + 1
print(count)        # 3
```

5.3 while loops

A `while` loop repeats **as long as** a condition stays true. Change something inside, or it never stops —an infinite loop 无限循环.

```
n = 1
while n <= 3:
    print(n)
    n = n + 1
# 1 2 3
```

- `break` leaves the loop straight away.

```
total = 0
while True:
    total = total + 10
    if total >= 30:
        break
print(total)        # 30
```

5.4 Nested loops

A loop inside another loop is a nested loop 嵌套循环. The inner loop 内层循环 runs fully for **each** turn of the outer loop 外层循环.

```
for row in range(3):
    line = ""
    for col in range(3):
        line = line + "*"
    print(line)
# ***
# ***
# ***
```

6 Lists & 2-D lists

6.1 Lists

A list 列表 holds many values in order, inside `[]`. Each item 元素 has an index (from 0).

```
scores = [88, 71, 95]
print(scores[0])      # 88
print(len(scores))    # 3
scores.append(60)     # add to the end
print(scores)         # [88, 71, 95, 60]
```

- Change an item by its index: `scores[1] = 100`.
- A list can grow and shrink; a string cannot.

6.2 Traversing a list

To traverse 遍历 a list is to visit each item. A `for` loop does this with no index needed.

```
scores = [88, 71, 95]
total = 0
for s in scores:
    total = total + s
print(total)          # 254
```

- Use `enumerate` when you also need the index.

```
for i, name in enumerate(["a", "b"]):
    print(i, name)    # 0 a  /  1 b
```

6.3 2-D lists (grids)

A 2-D list 二维列表 is a list of lists —a grid 网格 of rows and columns. Use two indexes: `grid[row][col]`.

```
grid = [[1, 2, 3],
        [4, 5, 6]]
print(grid[0][2])     # 3
print(grid[1][0])     # 4
```

- A nested loop 嵌套循环 visits every cell.

```
grid = [[1, 2], [3, 4]]
for row in grid:
    for value in row:
        print(value, end=" ")
print()               # 1 2 3 4
```

6.4 List comprehensions

A list comprehension 列表推导式 builds a new list in one line: `[expression for item in sequence]`.

```
squares = [x * x for x in range(5)]
print(squares)        # [0, 1, 4, 9, 16]
```

- Add `if` to keep only some items.

```
evens = [n for n in range(10) if n % 2 == 0]
print(evens)          # [0, 2, 4, 6, 8]
```

7 Dictionaries

7.1 Dictionaries

A dictionary 字典 (`dict`) stores key 键 → value 值 pairs. You look up a value by its key, not by a number index.

```
student = {"name": "Mei", "score": 88}
print(student["name"])     # Mei
print(student["score"])    # 88
```

7.1.1 Add and update

Assign to a key to add it, or to change an existing one.

```
student = {"name": "Mei"}
student["score"] = 88      # add a new key
student["score"] = 90      # update the value
print(student)             # {'name': 'Mei', 'score': 90}
```

7.1.2 Check and loop

Use `in` to test for a key. Loop over the keys, or over `.items()` to get both key and value.

```
student = {"name": "Mei", "score": 90}
print("score" in student)  # True
for key, value in student.items():
    print(key, "=", value)
# name = Mei
# score = 90
```

- `.get(key, default)` returns a default 默认值 when the key is missing —no error.

```
student = {"name": "Mei"}
print(student.get("age", 0))   # 0
```

8 Functions & abstraction

8.1 Defining & calling functions

A function 函数 is a named block of code you can reuse. Define 定义 it with `def`, then call 调用 it by name.

```
def greet():
    print("Hello!")

greet()        # Hello!
greet()        # Hello!
```

- The code inside runs only when you call the function.

8.2 Return values

A function can return 返回 a value with `return`. The call then stands for that value.

```
def square(n):
    return n * n

print(square(5))          # 25
print(square(3) + 1)      # 10
```

- `return` ends the function at once. A function with no `return` gives `None`.

8.3 Parameters, arguments & scope

A parameter 形参 is the name in the `def`. An argument 实参 is the value you pass in.

```
def power(base, exp):         # base, exp are parameters
    return base ** exp

print(power(2, 3))            # 8  (2 and 3 are arguments)
```

A variable made inside a function is local 局部—it exists only there. That region is its scope 作用域.

```
def f():
    x = 10            # local to f
    return x

print(f())            # 10
# print(x) here would be an error: x is not defined outside f
```

8.4 Procedural abstraction

Procedural abstraction 过程抽象 means hiding details behind a name. You use a function by its name and what it does, not by how it works.

```python
def area_of_rectangle(w, h):
    return w * h

print(area_of_rectangle(4, 5))   # 20
```

- A good function does one job, has a clear name, and avoids repeating code.

8.5 Modules & imports

A module 模块 is a file of ready-made functions. Bring one in with `import` 导入.

```python
import random
random.seed(0)                # makes the result repeatable
print(random.randint(1, 6))   # a dice roll

import math
print(math.sqrt(16))          # 4.0
```

9 Errors, exceptions & testing

9.1 Errors & debugging

Code can fail in three ways. A syntax error 语法错误 breaks Python's rules, so it never runs. A runtime error 运行时错误 happens while running, like dividing by zero. A logic error 逻辑错误 runs but gives the wrong answer.

```
# A runtime error, caught so this block still finishes:
try:
    print(10 / 0)
except ZeroDivisionError:
    print("cannot divide by zero")
# cannot divide by zero
```

- Python prints a traceback 回溯 showing where it failed. Read it from the bottom up.

9.2 try / except / raise

Wrap risky code in `try`. If it fails, `except` catches the exception 异常 and handles 处理 it, instead of crashing.

```
def to_int(text):
    try:
        return int(text)
    except ValueError:
        return 0

print(to_int("42"))    # 42
print(to_int("abc"))   # 0
```

- Catch a specific type (`ValueError`, `ZeroDivisionError`, ⋯).
- `raise` makes your own error on purpose.

```
def set_age(age):
    if age < 0:
        raise ValueError("age cannot be negative")
    return age

try:
    set_age(-1)
except ValueError as err:
    print("error:", err)
# error: age cannot be negative
```

9.3 Testing & robustness

A test 测试 checks that code gives the right answer. Try normal cases **and** edge cases 边界情形—empty input, zero, very large values.

```
def average(nums):
    if len(nums) == 0:        # edge case: empty list
        return 0
    return sum(nums) / len(nums)

print(average([2, 4, 6]))     # 4.0
print(average([]))            # 0
```

- Robust 健壮 code does not crash on strange input; it handles it gracefully.

10 Files

10.1 Reading & writing text files

A text file 文本文件 stores text on disk. Open it with `open(name, mode)` where mode 模式 says read or write. Always use `with`, which closes the file for you.

10.1.1 Writing

Mode `"w"` writes a new file and erases any old one.

```
with open("notes.txt", "w") as f:
    f.write("first line\n")
    f.write("second line\n")
print("saved")            # saved
```

10.1.2 Reading

Mode `"r"` (the default) reads. `.read()` returns the whole file as one string.

```
with open("notes.txt", "w") as f:
    f.write("hello\nworld\n")
with open("notes.txt") as f:
    print(f.read().strip())   # hello / world
```

10.1.3 Line by line

Loop over the file to get one line at a time. `.strip()` removes 去除 the newline 换行符 at the end.

```
with open("data.txt", "w") as f:
    f.write("Mei,88\nSam,71\n")
with open("data.txt") as f:
    for line in f:
        name, score = line.strip().split(",")
        print(name, "scored", score)
# Mei scored 88
# Sam scored 71
```

10.1.4 Appending

Mode `"a"` appends 追加—it adds to the end without erasing.

```
with open("log.txt", "w") as f:
    f.write("line 1\n")
with open("log.txt", "a") as f:
    f.write("line 2\n")
with open("log.txt") as f:
    print(f.read().strip())   # line 1 / line 2
```

Mode	Meaning
`"r"`	read (default)
`"w"`	write (erases first)
`"a"`	append (add to the end)

11 Algorithm design

11.1 Algorithms & decomposition

An algorithm 算法 is a clear list of steps that solves a problem. Decomposition 分解 means breaking a big problem into smaller parts you can solve one at a time.

```
# Algorithm: find the largest number in a list
def largest(nums):
    best = nums[0]
    for n in nums:
        if n > best:
            best = n
    return best

print(largest([3, 9, 2, 7]))   # 9
```

- Abstraction 抽象 means ignoring detail: you use `largest(...)` without re-reading how it works.

11.2 Pseudocode & flowcharts

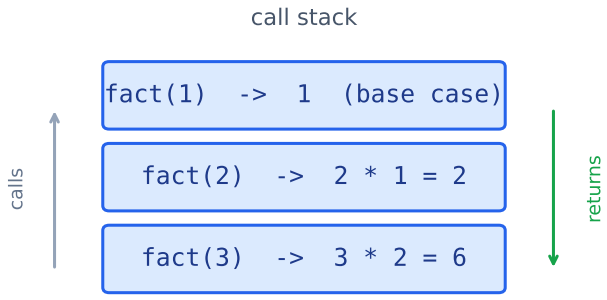
Pseudocode 伪代码 is plain, structured English for an algorithm, written before the real code. It is not run.

```
SET best TO first number
FOR each number n
    IF n > best THEN
        SET best TO n
OUTPUT best
```

A flowchart 流程图 draws the same plan: a box for each step, a diamond for each decision 判断, and arrows for the order.

11.3 Recursion & the call stack

Recursion 递归 is when a function calls itself. It needs a base case 基准情形 (a simple input that returns at once) and a recursive case 递归情形 (it calls itself on a smaller input).



The call stack for factorial(3): each call waits, then returns in reverse order

```
def fact(n):
    return 1 if n <= 1 else n * fact(n - 1)

print(fact(5))      # 120
```

- Each paused call sits on the call stack 调用栈 until the call above it returns.

12 Data structures

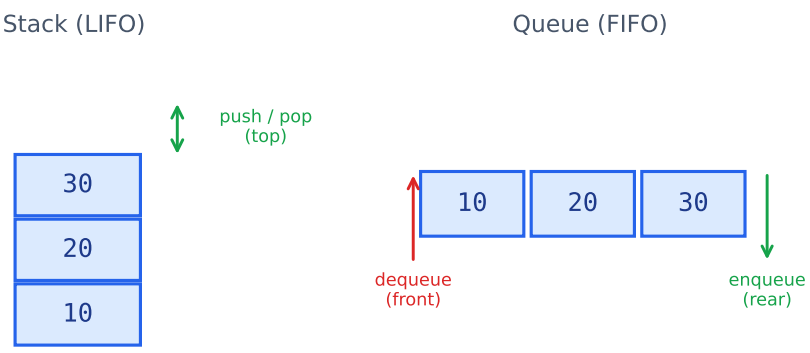
12.1 Abstract Data Types (ADTs)

An abstract data type 抽象数据类型 (ADT) describes some data plus the operations on it, separate from how it is built. You use it through its operations, not through its inner storage.

```
# A stack ADT, built on a list
s = []
s.append(1)        # add
s.append(2)
print(s.pop())     # 2  (remove the most recent)
```

12.2 Stacks

A stack 栈 is last-in, first-out (LIFO 后进先出). You push 压入 onto the top and pop 弹出 from the top.



A stack removes from the top (LIFO); a queue removes from the front (FIFO)

```
stack = []
stack.append("a")
stack.append("b")
print(stack.pop())   # b
print(stack.pop())   # a
```

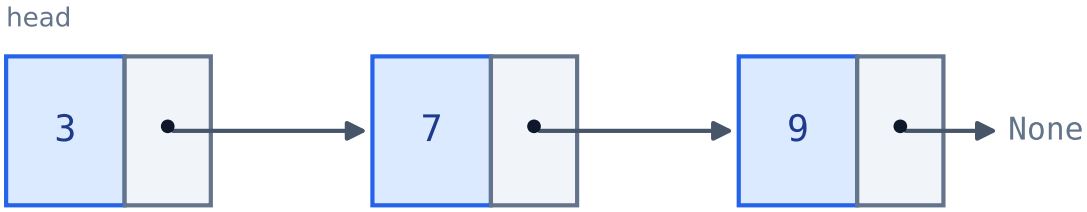
12.3 Queues

A queue 队列 is first-in, first-out (FIFO 先进先出). You enqueue 入队 at the back and dequeue 出队 from the front.

```
queue = []
queue.append("a")     # enqueue
queue.append("b")
print(queue.pop(0))   # a  (dequeue the front)
print(queue.pop(0))   # b
```

12.4 Linked lists

A linked list 链表 is a chain of nodes 节点. Each node holds data and a pointer 指针 to the next node; the last points to `None`.



A linked list: each node holds data and a pointer to the next node, ending at None

```
n3 = {"data": 3, "next": None}
n2 = {"data": 2, "next": n3}
n1 = {"data": 1, "next": n2}
node = n1
while node is not None:      # traverse to the end
    print(node["data"])
    node = node["next"]
# 1 2 3
```

12.5 Hash tables

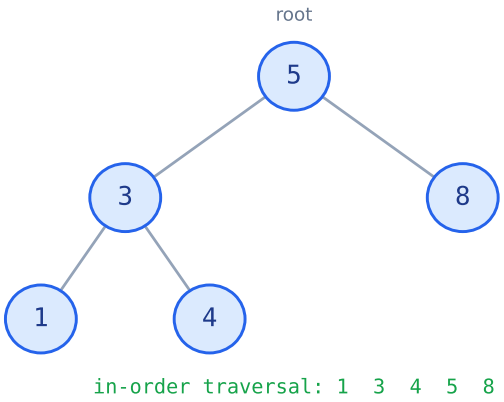
A hash table 哈希表 maps a key to a slot with a hash function 哈希函数. Two keys can land in the same slot —a collision 冲突. Python's `dict` is a hash table, so lookup is fast.

A hash function maps each key to a slot; two keys can collide in one slot

```
table = {}
table["Mei"] = 88
table["Sam"] = 71
print(table["Mei"])   # 88  (fast lookup by key)
```

12.6 Binary search trees

A binary search tree 二叉搜索树 (BST) keeps order: every left child is smaller than its node, every right child is larger. Search stays fast.

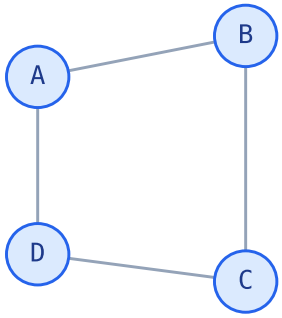


A binary search tree: smaller values go left, larger values go right

```
def insert(root, val):
    if root is None:
        return {"val": val, "left": None, "right": None}
    if val < root["val"]:
        root["left"] = insert(root["left"], val)
    else:
        root["right"] = insert(root["right"], val)
    return root

def inorder(root):
    if root is None:
        return []
    return inorder(root["left"]) + [root["val"]] + inorder(root["right"])

tree = None
for v in [5, 3, 8, 1, 4]:
    tree = insert(tree, v)
print(inorder(tree))   # [1, 3, 4, 5, 8]
```

12.7 Graphs

A graph 图 is a set of vertices 顶点 joined by edges 边. An adjacency list 邻接表—a dict of neighbour lists —is a common way to store one.




```
graph = {
  "A": ["B", "D"],
  "B": ["A", "C"],
  "C": ["B", "D"],
  "D": ["A", "C"],
}
```

A graph of vertices and edges, with its adjacency-list form

```
graph = {"A": ["B", "D"], "B": ["A", "C"], "C": ["B", "D"], "D": ["A",
↪  "C"]}
for vertex in graph:
    print(vertex, "->", graph[vertex])
# A -> ['B', 'D']  (and so on for B, C, D)
```

13 Searching, sorting & efficiency

13.1 Linear & binary search

A search 查找 finds where a value is. Linear search 线性查找 checks each item in turn, so it works on any list.

```
def linear_search(items, target):
    for i in range(len(items)):
        if items[i] == target:
            return i
    return -1     # not found

print(linear_search([4, 8, 2, 9], 2))   # 2
```

Binary search 二分查找 is much faster but needs a **sorted** list. It halves the range each step.

```
def binary_search(items, target):
    lo, hi = 0, len(items) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if items[mid] == target:
            return mid
        elif items[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1

print(binary_search([1, 3, 5, 7, 9], 7))   # 3
```

13.2 Sorting (bubble & insertion)

To sort 排序 is to put items in order. Bubble sort 冒泡排序 repeatedly swaps 交换 neighbours that are in the wrong order.

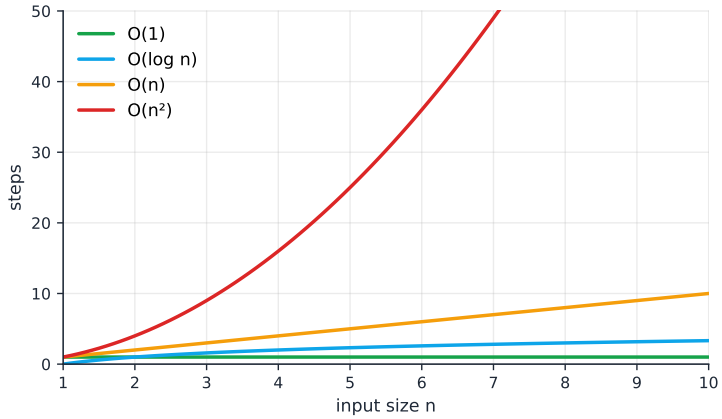
```
def bubble_sort(a):
    a = a[:]                          # work on a copy
    for i in range(len(a)):
        for j in range(len(a) - 1 - i):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
    return a

print(bubble_sort([5, 2, 4, 1]))      # [1, 2, 4, 5]
```

- Insertion sort 插入排序 builds a sorted part one item at a time.
- In real code, use Python's built-in `sorted()`:

```
print(sorted([5, 2, 4, 1]))          # [1, 2, 4, 5]
```

13.3 Algorithmic efficiency

Efficiency 效率 asks how the work grows as the input grows. We describe it with Big-O 大 O 记号.



How the number of steps grows with input size for common complexities

Big-O	Name	Example
$O(1)$	constant	look up a dict key
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	linear search
$O(n^2)$	quadratic	bubble sort

```
def steps(n):            # how many steps a linear scan takes
    count = 0
    for i in range(n):
        count = count + 1
    return count

print(steps(100))        # 100  -> O(n)
```

13.4 Randomness & simulation

The `random` module makes random numbers. Use a seed 种子 to make results repeatable. A simulation 模拟 runs many random trials to estimate an answer.

```
import random
random.seed(0)
rolls = [random.randint(1, 6) for _ in range(1000)]
print(rolls.count(6))   # about 1/6 of 1000
```

14 OOP & paradigms

14.1 Classes & objects

A class 类 is a blueprint. An object 对象 is one thing built from it (an instance 实例). `__init__` is the constructor 构造方法 that sets up each object; `self` is the object itself.

```
class Dog:
    def __init__(self, name):
        self.name = name            # an attribute
    def speak(self):
        return self.name + " says woof"

d = Dog("Rex")
print(d.speak())     # Rex says woof
```

- `name` is an attribute 属性 (data on the object); `speak` is a method 方法 (an action).

14.2 Inheritance, encapsulation & polymorphism

Inheritance 继承 lets a subclass 子类 reuse a superclass 父类. Use `super()` to call the parent; override 重写 a method to change it.

```
class Animal:
    def speak(self):
        return "some sound"

class Cat(Animal):
    def speak(self):                # override
        return "meow"

print(Cat().speak())   # meow
```

Encapsulation 封装 hides data behind methods; a leading underscore marks it private 私有.

```
class Account:
    def __init__(self):
        self._balance = 0           # private
    def deposit(self, n):
        self._balance += n
    def balance(self):
        return self._balance

a = Account()
a.deposit(50)
print(a.balance())     # 50
```

Polymorphism 多态 means one name, many behaviours —the right `speak` runs for each object.

```
class Cat:
    def speak(self):
        return "meow"

class Cow:
    def speak(self):
        return "moo"

for animal in [Cat(), Cow()]:
    print(animal.speak())   # meow, then moo
```

14.3 Programming paradigms

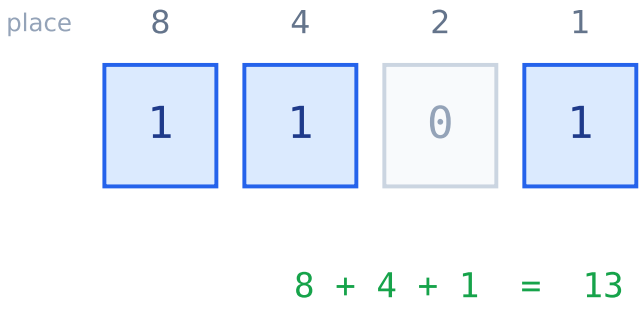
A paradigm 范式 is a style of writing programs. Procedural 过程式 code is a sequence of steps and functions. Object-oriented 面向对象 code groups data and methods into objects. Declarative 声明式 code says **what** you want, not how (a list comprehension or SQL).

```
def total(nums):           # procedural
    t = 0
    for n in nums:
        t += n
    return t
print(total([1, 2, 3]))    # 6

print(sum([1, 2, 3]))      # 6  (declarative: same result)
```

15 Data representation

15.1 Bits & binary

A bit 比特 is a single `0` or `1`. Binary 二进制 is the base-2 number system: each place is worth twice the one to its right (1, 2, 4, 8, ⋯). Denary 十进制 (base-10) is our normal numbers.



Binary place values: 1101 means 8 + 4 + 1 = 13

```
print(bin(13))          # 0b1101
print(int("1101", 2))   # 13
```

- 8 bits make a byte 字节. A fixed width can overflow 溢出 (wrap around) when the number is too big.

```
x = 250
x = (x + 10) % 256      # one byte wraps at 256
print(x)                # 4
```

15.2 Compression

Compression 压缩 makes data smaller. Lossless 无损 compression keeps every bit, so you rebuild the original exactly. Lossy 有损 compression throws away detail —smaller but not exact —and is used for photos and music.

Run-length encoding 游程编码 is a simple lossless method: store a run 游程 (a repeat) as a count plus the value.

```
def rle(text):
    out = ""
    i = 0
    while i < len(text):
        run = 1
        while i + run < len(text) and text[i + run] == text[i]:
            run += 1
        out += str(run) + text[i]
        i += run
```

```
    return out

print(rle("AAAABBBCCD"))   # 4A3B2C1D
```

16 Computing concepts

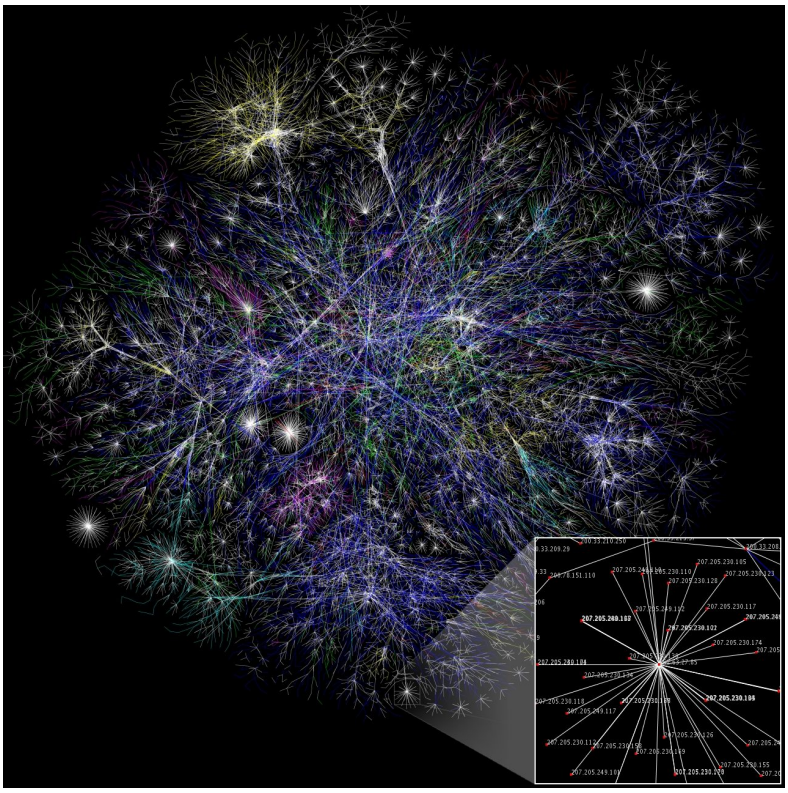
16.1 What computing is & the design cycle

Computing 计算 means solving problems with computers: input, process, output. Good software is built in a design cycle 设计循环—plan, write, test, improve —repeated many times.

- Break a problem down, build a small part, test it, then add more.
- Programmers work in teams and reuse each other's code.

16.2 The Internet

The Internet 互联网 is a network 网络 of networks. Data is split into packets 数据包 that travel separately and are put back together at the other end. Shared rules called protocols 协议 (such as TCP/IP) make this work. If one path breaks, packets take another route —this is redundancy 冗余, which gives fault tolerance 容错.



A map of the Internet: each line is a path between two networks

Image: The Opte Project, CC BY 2.5 (commons.wikimedia.org)

Layer	Job
HTTP	request and send web pages
TCP	reliable delivery, in order
IP	addressing and routing

16.3 Parallel & distributed computing

Sequential 顺序 code does one step at a time. Parallel 并行 computing does several steps at once on many cores 核心, which can give a speedup 加速. Distributed 分布式 computing spreads the work across many computers, such as a cloud.

- Not everything can run in parallel: some steps must wait for an earlier result.

16.4 Impact of computing

Computing brings both benefits and harms. The digital divide 数字鸿沟 means not everyone has equal access to it. Software can carry bias 偏见 from the data it learns from. Respect intellectual property 知识产权 (licences), and protect people's personal data 个人数据 and privacy 隐私.

17 Putting it together

17.1 End-to-end mini-projects

A mini-project 小项目 combines earlier ideas: data in a list, a function with selection inside a loop, and printed output. This is also the shape of the AP Create Performance Task.

17.1.1 Project: average mark

```
def average(marks):
    if len(marks) == 0:
        return 0
    return round(sum(marks) / len(marks), 1)

print(average([88, 71, 95, 60]))   # 78.5
```

17.1.2 Project: count passes

```
def count_passes(marks, pass_mark=60):
    passes = 0
    for m in marks:                 # iteration
        if m >= pass_mark:          # selection
            passes += 1
    return passes

print(count_passes([88, 50, 95, 60]))   # 3
```

17.1.3 Project: filter to a new list

```
def merit(marks):
    return [m for m in marks if m >= 80]

print(merit([88, 71, 95, 60]))     # [88, 95]
```

The AP Create Task wants a list, a parameterised procedure 过程 that uses selection 选择 and iteration 迭代, and some input/output. Each project above is exactly that shape—build small pieces, then join them.